Jalan-Jalan Bahagia dengan Menerapkan *Dynamic Programming*

Dharma Kurnia Septialoka - 13514028[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13514028@std.stei.itb.ac.id

***Abstract*—Setiap orang mencintai *traveling.* Bepergian dan mengeksplorasi tempat baru adalah hal yang asyik. Namun seringkali saat ingin bepergian, kita sulit menentukan mana barang yang seharusnya kita bawa dan tidak untuk tas atau koper kita. Saat kita berada di penginapan pun, kita ingin sering kali menjelajahi kota tersebut dengan sebisa mungkin mengunjungi tempat-tempat menarik di kota tersebut tanpa harus melewati jalan yang tidak perlu berulang-ulang. Dalam makalah ini, akan diimplementasikan pemilihan barang teroptimal untuk tas dan jalur eksplorasi kota yang paling efisien dengan memanfaatkan *dynamic programming.* Pada bagian awal, akan dibahas mengenai traveling secara singkat, fakta-fakta, dan latar belakang pemilihan permasalahan ini. Lalu, akan dijelaskan dasar teori dari *dynamic programming* itu sendiri dan bagaimana implementasinya pada program jalan-jalan bahagia kali ini.**

***Index Terms*—Dynamic Programming, Knapsack, Traveling, Tour**

I. PENDAHULUAN

Traveling atau jalan-jalan adalah hobi yang mengasyikkan. [1]*Traveling* adalah aktivitas melancong; berpindah dari satu tempat ke tempat lainnya karena berbagai alasan seperti pekerjaan, liburan, dan lainnya. *Traveling* adalah cara untuk membuka wawasan dan memperluas pengetahuan dalam mengunjungi tempat baru atau tempat yang sudah pernah dikunjungi sebelumnya dengan berinteraksi pada objek disekitarnya. Saat ingin bepergian, seringkali bagasi kita hanya diberi jatah 15 kg dan perlu membayar *charge* atau sejumlah uang atas berat yang kelebihan. Barang yang ingin kita bawa sangat banyak, karena yang tersedia sangat banyak. Kita pun tau berat dari setiap jenis barang yang ingin kita bawa, dan kita dapat menentukan skala pentingnya barang tersebut untuk dibawa menurut kita. Tapi seringkali kita hanya memanfaatkan intuisi kita atas barang apa yang perlu dibawa dan yang tidak, yang akhirnya sering menimbulkan penyesalan di akhir. Begitupula saat kita sudah sampai di tempat penginapan, kita akan bepergian ke tempat-tempat baru dengan membawa tas ransel, yang tentu saja ada bobot maksimum agar kita tetap nyaman dalam berjalan-jalan dan agar tas kita tidak rusak. Kita pun sering bingung menentukan barang apa yang sebaiknya kita bawa dan yang tidak. Karena sangat bingungnya, banyak sekali artikel-artikel bertebaran yang berisi barang-barang yang sebaiknya kamu perlu bawa dan tidak saat traveling. *Search recommendation by google* sangat banyak untuk *keyword* seperti 'barang penting untuk travel', 'list barang untuk travel', 'barang yang tidak perlu dibawa saat liburan', dan sebagainya. Menurut saya, artikel seperti itu kurang efisien, karena tentu kebutuhan setiap orang berbeda-beda, sehingga diperlukan suatu *tools* yang dapat membantu traveler menentukan barang apa yang perlu dibawa dan tidak saat *occasion* tertentu.

Begitu juga, saat kita sampai disuatu tempat, seringkali kita ingin mengeksplorasi kota itu dengan mengunjungi sebanyak mungkin tempat-tempat menarik yang para turis biasa kunjungi. Tetapi, tentu jika kita mengunjungi semuanya, kita akan menghabiskan banyak waktu karena pasti akan terdapat banyak jalur sama yang kita lewati. Belum lagi jika ada titik-titik lokasi yang merupakan *blocked area,* atau tidak bisa dilewati, atau tidak ingin kita lewati (karena sudah pernah dan sebagainya), persoalan ini jadi semakin rumit. Karena tujuan kita ingin mengeksplorasi kota dan mengunjungi tempat menarik sebanyak mungkin, maka kita akan pergi ke ujung kota lalu kembali ke tempat semula yakni penginapan kita. Misalkan saja, hotel kita berada di utara-barat (*top left corner)* dan kita akan pergi ke ujung kota yakni yang berada di selatan-timur (*bottom right* corner). Karena kita malas dan untuk penyederhanaan, maka saat kita pergi kita hanya akan menuju selatan atau timur peta, dan saat kembali pulang hanya akan menuju utara atau barat peta. Persoalannya adalah bagaimana agar perjalanan kita efisien dan dapat memaksimalkan tujuan kita yakni mengunjungi sebanyak mungkin lokasi menarik tanpa melewati

blocked area. Permasalahan ini terinspirasi dari *Manhattan Tourist Problem* dan *problem* yang dilihat pada *Sphere Online Judge* dengan sedikit modifikasi[2].

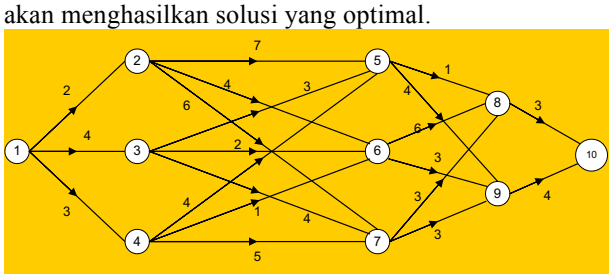
Karena melihat dua permasalahan pada jalan-jalan inilah, penulis terinspirasi untuk menerapkan program jalan-jalan bahagia dengan memanfaatkan *dynamic programming*.

II. LANDASAN TEORI *DYNAMIC PROGRAMMING*

Algoritma pemrograman dinamis (*dynamic programming*) merupakan metode pemecahan masalah dengan cara menguraikan solusi menjadi sekumpulan langkah (step) atau tahapan (stage) sedemikian sehingga solusi dari persoalan dapat dipandang dari serangkaian keputusan yang saling berkaitan[3]. Rangkaian keputusan ini akan membuat solusi untuk permasalahan yang ada.

Syarat pertama agar suatu permasalahan dapat diselesaikan dengan program dinamis adalah permasalahan tersebut harus dapat dipecah-pecah menjadi keputusan-keputusan dengan jumlah yang terbatas. Kemudian suatu permasalahan harus dapat dibagi berdasarkan prinsip optimalitas dimana permasalahan tersebut dapat dipecah menjadi masalah-masalah yang lebih kecil untuk kemudian dicari solusi optimalnya. Solusi optimal untuk setiap masalah ini kemudian akan menjadi dasar untuk solusi optimal dari permasalahan yang ada.

Solusi untuk setiap upa-masalah mungkin ada lebih dari satu, oleh karena itu dalam mendapatkan solusi optimal harus dipertimbangkan berbagai solusi yang ada. Maka untuk setiap langkah program dinamis akan mempertimbangkan beberapa rangkaian keputusan, pada akhirnya program dinamis akan mempertimbangkan seluruh kemungkinan solusi yang ada. Hal ini membedakan program dinamis dengan greedy dalam memecahkan masalah, dimana greedy hanya mempertimbangkan satu kemungkinan solusi. Hasil dari program dinamis juga berbeda dengan greedy, dalam hal ini greedy dapat menghasilkan solusi yang optimal dan dapat juga menghasilkan solusi yang kurang optimal sedangkan program dinamis dipastikan akan menghasilkan solusi yang optimal.



Gambar 1 – Graf Multitahap dari Munir,Rinaldi. Slide Kuliah Program Dinamis. 2015

Terdapat dua solusi pendekatan programa dinamis, yaitu pendekatan maju (forward atau up-down) dan pendekatan mundur (backward atau bottom-up).

1. Program dinamis maju.
Program dinamis bergerak mulai dari tahap 1, terus maju ke tahap 2, 3, dan seterusnya sampai tahap n. Runtunan peubah keputusan adalah x1, x2, …, xn.

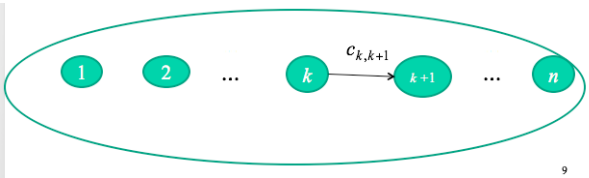
2. Program dinamis mundur.
Program dinamis bergerak mulai dari tahap n, terus mundur ke tahap n – 1, n – 2, dan seterusnya sampai tahap 1. Runtunan peubah keputusan adalah xn, xn-1, …, x1.

Perbedaan Algoritma Greedy dengan Program Dinamis:

- Greedy: hanya satu rangkaian keputusan yang dihasilkan
- Program dinamis: lebih dari satu rangkaian keputusan yang dipertimbangkan.

Keuntungan lain dari program dinamis adalah program dinamis menyimpan hasil untuk semua upa-masalah, sehingga jika terdapat upa-masalah yang overlap maka upa-masalah tersebut tidak perlu dicari kembali solusi optimalnya.

Pada program dinamis, rangkaian keputusan yang optimal dibuat dengan menggunakan **Prinsip Optimalitas**. Prinsip Optimalitas: *jika solusi total optimal, maka bagian solusi sampai tahap ke-k juga optimal*. Prinsip optimalitas berarti bahwa jika kita bekerja dari tahap k ke tahap $k + 1$, kita dapat menggunakan hasil optimal dari tahap k tanpa harus kembali ke tahap awal. Ongkos pada tahap k +1 = (ongkos yang dihasilkan pada tahap k) + (ongkos dari tahap k ke tahap $k + 1$)



Gambar 2 - Prinsip Optimalitas dari Munir,Rinaldi. Slide Kuliah Program Dinamis. 2015

Berikut karakteristik Persoalan Program Dinamis:

1. Persoalan dapat dibagi menjadi beberapa tahap (stage), yang pada setiap tahap hanya diambil satu keputusan.

2. Masing-masing tahap terdiri dari sejumlah status (state) yang berhubungan dengan tahap tersebut. Secara umum, status merupakan bermacam kemungkinan masukan yang ada pada tahap tersebut.

3. Hasil dari keputusan yang diambil pada setiap tahap ditransformasikan dari status yang bersangkutan ke status berikutnya pada tahap berikutnya.

4. Ongkos (cost) pada suatu tahap meningkat secara teratur (steadily) dengan bertambahnya jumlah tahapan.

5. Ongkos pada suatu tahap bergantung pada ongkos tahap-tahap yang sudah berjalan dan ongkos pada tahap tersebut.

6. Keputusan terbaik pada suatu tahap bersifat independen terhadap keputusan yang dilakukan pada tahap sebelumnya.

7. Adanya hubungan rekursif yang mengidentifikasikan keputusan terbaik untuk setiap status pada tahap k memberikan keputusan terbaik untuk setiap status pada tahap k + 1.

8. Prinsip optimalitas berlaku pada persoalan tersebut.

Secara umum, ada empat langkah yang dilakukan dalam mengembangkan algoritma pemrograman dinamis:

1. Karakteristikkan struktur solusi optimal

2. Definisikan secara rekursif nilai solusi optimal

3. Hitung nilai solusi optimal secara maju atau mundur

4. Konstruksi solusi optimal

III. PENERAPAN *DYNAMIC PROGRAMMING*

Tampilan awal program jalan-jalan bahagia adalah sebagai berikut:

```
Dharma-Kurnia-Septialokas-MacBook-Pro:makalah septialoka$ ./a
Selamat datang di program jalan-jalan bahagia
Silahkan pilih menu dibawah untuk membantu perjalananmu!

1. Membantu kamu memilih barang yang akan dimasukkan ke tas kamu
2. Membantu kamu memilih jalur tur jalan-jalan kamu agar lebih efisien
3. Exit program
```

Gambar 3 – Tampilan awal program

- Optimalisasi Tas Bawaan

Untuk persoalan kali ini, sebenarnya mirip dengan persoalan knapsack 0/1 yang diimplementasikan pada program jalan-jalan bahagia. Intinya adalah dengan menggunakan program dinamis untuk menentukan apakah barang tersebut akan dimasukkan ke tas atau tidak. Kita akan mendekomposisi permasalah ini menjadi tahap dan status dimana:

1. Tahap (k) adalah proses memasukkan barang ke tas
2. Status(y) merupakan kapasitas muat tas yang tersisa setelah memasukkan barang pada tahap sebelumnya

Dari tahap ke-1, kita memasukkan objek ke-1 kedalam tas untuk setiap satuan kapasitas tas sampai batas kapasitas maksimumnya. Misalkan ketika memasukkan objek pada tahap k, kapasitas muat tas sekarang adalah $y - w_k$. Untuk mengisi kapasitas sisanya, kita menerapkan prinsip optimalitas dengan mengacu pada nilai optimum dari tahap sebelumnya untuk kapasitas sisa $y - w_k$ (yaitu $f_{k-1}(y - w_k)$).

Selanjutnya, kita bandingkan nilai keuntungan dari objek pada tahap k (yaitu pk) plus nilai $f_{k-1}(y - w_k)$ dengan keuntungan pengisian hanya $k - 1$ macam objek, $f_{k-1}(y)$.

Jika $p_k + f_{k-1}(y - w_k)$ lebih kecil dari $f_{k-1}(y)$, maka objek yang ke-k tidak dimasukkan ke dalam tas, tetapi jika lebih besar, maka objek yang ke-k dimasukkan.

Relasi rekurens untuk persoalan ini adalah

$f_0(y) = 0, \quad y = 0, 1, 2, \ldots, M$ (basis)

$f_k(y) = -\infty, \quad y < 0$ (basis)

$f_k(y) = \max\{f_{k-1}(y), p_k + f_{k-1}(y - w_k)\}$, (rekurens)

$k = 1, 2, \ldots, n$

$f_k(y)$ adalah keuntungan optimum dari persoalan 0/1 *Knapsack* pada tahap k untuk kapasitas tas sebesar y.

$f_0(y) = 0$ adalah nilai dari persoalan *knapsack* kosong (tidak ada persoalan *knapscak*) dengan kapasitas y,

$f_k(y) = -\infty$ adalah nilai dari persoalan knapsack untuk kapasitas negatif. Solusi optimum dari persoalan 0/1 *Knapsack* adalah $f_n(M)$.

Implementasi-nya adalah sebagai berikut:
Pada program, dibuat tipe bentukan sebagai berikut:

```
typedef struct {
    string namabarang;
    int berat;
    int nilai;
} item;

vector<item> listitem;
```

Gambar 4 – Tipe bentukan item

Dengan satu fungsi misalkan bernama knapsack untuk mengembalikan nilai maksimum pada barang yang akan dimasukkan tas, yaitu:

```
Function knapsack (input listitem:
vector of item, input/output
bestItems: set of integer, input
jumlahbarang: integer, input
maxweight: integer)→ integer

Kamus
bestValues =
        array[jumlahbarang][maxweight]
        of integer

Algoritma
for (w= 0 to maxweight)
```

```
        bestValues[0, w]← 0
for (i = 1 to jumlahbarang)
    for (w= 0 to maxweight)
        if ((listitem[i].berat  w) and
            (listitem[i].nilai +
            bestValue[i-1, w-
            listitem[i].berat] >
            bestValues[i-1, w]))
        {
            bestValues[i, w] ←
                listitem[i].value +
                bestValues[i-1, w -
                listitem[i].berat]
            keep[i, w] ← 1
        }
        else
        {
            bestValues[i, w] ←
                bestValues[i-1, w]
            keep[i, w] ← 0
        }
K ← maxweight
for (i = n downto 1)
     if (keep[i, k] = 1)
     {
        bestItems.insert(i)
        K ← K - listitem[i].berat
     }
→ bestValues[n, maxweight]
```

Berikut adalah implementasi program utama:

```
Program Utama

Input(jumlahbarang)
Input(maxweight)
for (i= 1 to jumlahbarang)
        Input (barang, berat, dan
                nilai)
        push ke vector listitem
Output("Berikut adalah barang yang
sebaiknya kamu bawa:")
Totalberat ← 0
Totalnilai ← knapsack(listitem,
  bestItems, jumlahbarang, maxweight)
for (iterator it = bestItems.begin()
to bestItems.end())
      Output(listitem[it].namabarang)
      Totalberat ← Totalberat +
                listitem[it].berat
endfor
Output("Total: ", Totalberat,
        Totalnilai)
```

Tampilan program saat meminta input user:

```
Silahkan masukkan berapa banyak barang yang tersedia yang
amu ingin bawa:

Jumlah barang yg ada = 18
Silahkan masukkan berat maksimum untuk tasmu!
Berat maksimum yg kamu perbolehkan = 4000

Masukkan 1 per 1 barangmu, beserta beratnya dalam gram dan
nilai relatif barang tersebut menurut kamu (seberapa penti
g barang tersebut untuk dibawa dari skala 1-100

Nama barang ke-1 = botol minum
Berat barang ke -1 = 700
Nilai barang ke -1 = 90

Nama barang ke-2 = buku tulis
Berat barang ke -2 = 300
Nilai barang ke -2 = 30

Nama barang ke-3 =
```

Gambar 5 – Tampilan input barang yang tersedia

Contoh data uji 1 adalah sebagai berikut:

No	Nama Barang	Berat (gram)	Nilai
1	Botol minum	1200	90
2	Buku tulis	300	25
3	Kotak pensil	500	20
4	Dompet	400	90
5	Roti	200	50
6	Payung	600	60
7	Komik conan	300	10
8	Baju ganti	200	40
9	Tissue	300	70
10	Celana ganti	800	20
11	Kamera SLR	1400	50
12	Handuk kecil	500	40
13	Bekal nasi	1000	40
14	Tripod	400	60
15	Sandwich	200	20
16	Pizza	600	60
17	Kacamata gaul	200	70
18	iPad	700	60

Tabel 1 – Data uji 1 input barang

Berikut keluaran program tentang barang yang disarankan untuk dibawa dengan data uji 1:

```
Berikut adalah barang yang sebaiknya kamu bawa:
buku tulis                300    25
dompet                    400    90
roti                      200    50
payung                    600    60
baju ganti                200    40
tissue                    300    70
handuk kecil              500    40
tripod                    400    60
sandwich                  200    20
pizza                     600    60
kacamata gaul             200    70
Total:                   3900   585
Dharma-Kurnia-Septialokas-MacBook-Pro:makalah se
```

Gambar 6 – Keluaran program untuk data uji 1

Contoh data uji 2:

No	Nama Barang	Berat (gram)	Nilai
1	Botol minum	700	90
2	Buku tulis	300	30
3	Kotak pensil	500	20
4	Dompet	300	90
5	Roti	200	40
6	Payung	600	60
7	Komik conan	300	30
8	Baju ganti	200	50
9	Tissue	300	60
10	Celana ganti	800	30
11	Kamera SLR	1400	40
12	Handuk kecil	500	40
13	Bekal nasi	1000	50
14	Tripod	400	40
15	Sandwich	200	20
16	Pizza	600	60
17	Kacamata gaul	200	60
18	iPad	700	40

Tabel 2 – Data uji 2 input barang

Berikut keluaran program tentang barang yang disarankan untuk dibawa dengan data uji 2:

```
Berikut adalah barang yang sebaiknya kamu bawa:
botol minum               700    90
buku tulis                300    30
dompet                    300    90
roti                      200    40
payung                    600    60
baju ganti                200    50
tissue                    300    60
tripod                    400    40
sandwich                  200    20
pizza                     600    60
kacamata gaul             200    60
Total:                   4000   600
Dharma-Kurnia-Septialokas-MacBook-Pro:makalah sep
```

Gambar 7 – Keluaran program untuk data uji 2

Dapat kita lihat dari function knapsack bahwa kompleksitas untuk program ini adalah O(nW) dengan n adalah jumlah barang yang tersedia dan W adalah berat maksimum untuk tas yang diperbolehkan. Pada program ini, yang kita lakukan adalah bagaimana memaksimalkan *value* dengan jumlah beratnya tidak melebihi batas berat maksimum. Pertama-tama, kita menginisialisasi bestValue[0, w] dengan 0 lalu untuk rekursifnya yaitu bestValue[i, w] dengan nilai maks dari (bestValue[i-1, w] dan listitem[i].value + bestValue[i-1, w - listitem[i].berat] dengan $1 \leq i \leq$ jumlahbarang, $0 \leq w \leq$ maxweight.

- Optimalisasi Jalur Tur Jalan-Jalan

Seringkali saat kita bepergian atau berjalan-jalan ke suatu tempat, *traveling,* dan semacamnya, kita banyak membuang-buang waktu untuk mengelilingi atau melewati jalan yang sebetulnya tidak kita perlukan. Saat kita mengunjungi suatu tempat baru, kita ingin sebisa mungkin tidak melewati lokasi yang sama dua kali dan sebisa mungkin mengunjungi sebanyak mungkin tempat-tempat yang menarik. Tetapi sebagai turis, tentu kita malas jika perlu menghafal banyak jalan, dan ingin menghemat waktu karena kita akan sering berpindah-pindah kota, jadi kita pun ingin agar arah jalannya tidak terlalu sulit. Untuk penyederhanaan, misalkan saja kita berada di suatu penginapan yang lokasinya di kota bagian utara-barat, maka untuk mengeksplorasi kota tersebut, kita akan pergi ke bagian selatan-timur dengan hanya berjalan ke timur atau ke selatan, dan saat kembali, kita hanya akan berjalan ke utara atau ke barat. Tentu permasalahan ini mudah, tapi seperti di dunia nyata, misalkan saja ada tempat-tempat yang diblokir atau tidak bisa dilewati, karena sulit diakses, dan semacamnya. Maka, bagaimana kita bisa tahu berapa banyak tempat menarik yang dapat kita kunjungi?

Tampilan program saat pengguna memilih pilihan 2:
Data Uji 1:

```
Start akan selalu dimulai dari utara-barat, lalu silahkan tandai
tempat menarik dengan huruf abjad, lalu area yang tidak bisa kam
 lewati dengan #
Masukkan lebar kotamu dalam satuan = 5
Masukkan panjang kotamu dalam satuan = 5
.a.b.
c###.
d.e.f
.###g
.h.i.

Area yang kamu sebaiknya lewati = c d h i g f b a
Tempat menarik maksimal yang dapat kamu kunjungi sebanyak 8
Dharma-Kurnia-Septialokas-MacBook-Pro:makalah septialoka$
```

Gambar 8 – Data uji 1 input peta

Data Uji 2:

```
Start akan selalu dimulai dari utara-barat, lalu silahkan tandai
tempat menarik dengan huruf abjad, lalu area yang tidak bisa kamu
 lewati dengan #
Masukkan lebar kotamu dalam satuan = 3
Masukkan panjang kotamu dalam satuan = 3
.#a
.#b
...

Area yang kamu sebaiknya lewati =
Tempat menarik maksimal yang dapat kamu kunjungi sebanyak 0
Dharma-Kurnia-Septialokas-MacBook-Pro:makalah septialoka$
```

Gambar 9 – Data uji 2 input peta

Asumsikan sebagai pengguna telah mengetahui peta kota secara umum. Pengguna meng-input '.' untuk lokasi yang dapat dilewati, huruf abjad untuk inisial nama lokasi menarik, dan '#' untuk tempat yang tidak dapat dilewati. Pada data uji 1, tur akan dimulai dari cdhigfba, sedangkan pada data uji 2, pengguna tidak akan bisa menuju lokasi menarik karena terdapat area buntu. Implementasi programnya dengan memanggil satu prosedur sebagai berikut:

```
procedure turWithDP (input H:
integer, input W: integer, input
board: array [100] of string,
input/output dp: array[200][100][100]
of integer, input/output tempat:
array[200][100][100] of string)

Kamus
        -
Algoritma
if (board[0][0] adalah char)
        dp[0][0][0] ← 1
        tempat[0][0][0] ← char
                tersebut
for(k=1 to H+W-2)
  for(i=0 to W-1)
     for(j=0 to W-1)
        dp[k][i][j] ← -∞;
        if not (i>k or j>k or k-i
              ≥ H or k-j ≥ H) or not
              (board[k-i][i]= '#' or
               board[k-j][j] = '#')
           if(k-1-i ≥ 0 and k-1-j
                        ≥ 0)
              dp[k][i][j] ← dp[k-
                        1][i][j];
              tempat[k][i][j] ←
                tempat[k-1][i][j]
           if(i > 0 && k-1-j ≥ 0)
              dp[k][i][j] ←
                max(dp[k][i][j], dp[k-
                        1][i-1][j]);
            if (dp[k-1][i-1][j] >
              dp[k][i][j])
                tempat[k][i][j] ←
                  tempat[k-1][i-
                        1][j];
          if(j > 0 && k-1-i >= 0)
            dp[k][i][j] ←
              max(dp[k][i][j], dp[k-
              1][i][j-1]);
          if (dp[k-1][i][j-1] >
                        dp[k][i][j])
            tempat[k][i][j] ←
              tempat[k-1][i][j-1];
          if(i > 0 and j > 0)
             dp[k][i][j] ←
               max(dp[k][i][j], dp[k-
                        1][i-1][j-1])
          if (dp[k-1][i-1][j-1] >
                        dp[k][i][j])
             tempat[k][i][j] ←
               tempat[k-1][i-1][j-1]
          if(i = j)
            if((board[k-i][i]) adalah
            char)
              increment dp[k][i][j]
              tempat[k][i][j] ←
                tempat[k][i][j] +
                board[k-i][i]
          else if((board[k-i][i])
          adalah char)
              increment dp[k][i][j]
              tempat[k][i][j] ←
                tempat[k][i][j] +
                board[k-i][i]
          else if(board[k-j][j])
          adalah char)
              increment dp[k][i][j]
              tempat[k][i][j] ←
                tempat[k][i][j +
                board[k-j][j]
```

Untuk program utamanya sendiri, maka implementasinya adalah sebagai berikut:

```
Program Utama

input(panjang kota)
input(lebar kota)
for (i=1 to panjangkota)
input(board[i])
inisialisasi tempat[][][] dengan
string kosong
if (board[0][0] adalah char)
        dp[0][0][0] <- 1
        tempat[0][0][0] <-
          tempat[0][0][0] + char tsb
else
        dp[0][0][0] <- 0
turWithDP (panjangkota, lebarkota,
board, dp, tempat)
output("Area yang kamu sebaiknya
  lewati = ", tempat[H+W-2][W-1][W-
  1])
output("Tempat menarik maksimal yang
  dapat kamu kunjungi sebanyak ",
  dp[H+W-2][W-1][W-1])
```

Dapat kita lihat bahwa jika kita menganggap tur ini independen pada jalur pergi dan pulang, maka saat backtrack kita akan banyak melewati jalur yang sama, jalan pulang sangat bergantung dengan jalan pergi. Maka itu, kita perlu menghitungnya secara simultan. Gampangnya, kita akan menganggap permasalahan ini dengan membayangkan dua orang, mulai dari lokasi (0,0) dan (0,0) sehingga akan ada 4 kemungkinan perpindahan (dua kemungkinan untuk setiap orang). Saat jalan pulang, pengguna hanya akan dapat ke kiri atau ke atas (barat-utara), begitupula sama dengan saat jalan pergi yaitu hanya dapat ke kanan atau ke bawah (selatan-timur). Jadi sebenarnya tidak ada bedanya antara pergi dan pulang. Karena sama, maka misalkan saja kita ambil kasus saat pergi, dua orang akan pergi mulai dari titik (0,0) dan (0,0). Sehingga lokasi yang kita analisa akan ada 4 titik yakni x1, y1, x2, y2. Jika dua orang tersebut berada di titik yang sama, maka kita akan *increment* DP jika pada *board* merupakan lokasi yang menarik. Kompleksitas untuk turWithDP adalah $O(n^3)$ karena x1 + y1 = x2 + y2 (kita tahu bahwa jarak yang ditempuh pasti sama saat waktunya sama),

sehingga y2 = x1+y1-x2 sehingga kita hanya akan melihat kondisi (x1, y1, x2).

Langkah penyelesaiannya adalah sebagai berikut[4]:

1. Kita akan melihat permasalahan ini seperti mencari 2 jalur dari utara-barat ke selatan-timur dengan memaksimalkan jumlah alphabet (tempat-tempat yang menarik)
2. Karena jarak jalan yang ditempuh tetap, yakni H+W-2, apapun pergerakan kita, maka bayangkan untuk memperluas 2 jalur tersebut secara bersamaan.
3. Misal DP(k, I, j) adalah jumlah maksimum alphabet yang ada pada jalur dari (0,0) ke (k-I, i) dan (k-j, j). Setiap kali bergerak, maka increment k, dan untuk setiap i, j, kita akan meng-increment nilai DP
4. Untuk setiap tahap, kita memiliki 4 kemungkinan. Maka nilai DP(k, i, j) adalah maksimum dari {DP(k-1, i-1, j), DP(k-1, i, j-1), DP(k-1, i-1, j-1), DP(k-1, i, j)} + {0, 1, atau 2 bergantung apakah (k-i,i) dan (k-j,j) pada board merupakan alfabet. Jika salah satu dari (k-i,i) atau (k-j,j) merupakan tempat yang tidak bisa dilewati ('#'), maka kita set DP(k, i, j) = $-\infty$

V. SIMPULAN

Dynamic Programming sangat baik diaplikasikan untuk permasalahan yang membutuhkan optimalitas. Perbedaan dengan algoritma *greedy* yang sangat mencolok adalah DP tetap mempertimbangkan subsolusi sebelum dan sesudah untuk dipertimbangkan kembali sehingga seringkali menawarkan lebih dari 1 rangkaian keputusan. DP juga memecah masalah menjadi upamasalah yang lebih kecil sehingga subsolusi juga lebih mudah untuk di-*maintain* dan di-*debug*. Untuk masalah kompleksitas, Greedy tentu lebih efisien dari DP. Dengan memanfaatkan DP pada program jalan-jalan bahagia, kita dapat dengan mudah mengetahui barang apa yang perlu kita bawa dan tidak, juga kita dengan mudah menentukan rute terbaik saat mengeksplorasi suatu kota.

VI. UCAPAN TERIMA KASIH

Penulis pertama-tama ingin mengucapkan terima kasih kepada orang tua penulis, karena sampai detik ini ayah dan ibu selalu membimbing, mendidik, dan mendukung penulis, baik secara moril maupun non-moril. Penulis juga mengucapkan terima kasih yang sebesar-besarnya kepada Pak Rinaldi Munir dan Bu Nur Ulfa M atas pengajarannya selama ini pada mata kuliah Strategi Algoritma, terutama pada materi *dynamic programming*. Terakhir, penulis juga ingin bersyukur dan mengucapkan terima kasih kepada keluarga, teman, lingkungan, dosen-dosen lain, dan semuanya.

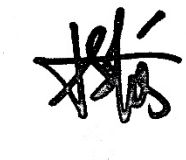
REFERENSI

[1] http://www.qorisme.com/2013/02/artikel-apa-itu-traveling.html
[2] http://www.spoj.com/problems/TOURIST/
[3] Munir,Rinaldi. Slide Kuliah Program Dinamis. 2015. Situs : http://informatika.stei.itb.ac.id/~rinaldi.munir /Stmik/stmik.htm
[4] http://abitofcs.blogspot.co.id/2014/12/spoj-tourist-tourist.html

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 5 Mei 2016



Dharma Kurnia Septialoka – 13514028